A Design Theory for Fast and Slow Thinking in Interactive Coding

Evan S. Raskob
The CCI at UAL
evan.raskob@arts.ac.uk

ABSTRACT

This paper introduces a new theory and shared language to help users and designers of interactive programming (IP) systems, especially those aimed at producing "creative" multimedia outcomes such as generative sculpture, art, music, graphics, and livecoding performances, to better identify key programming activities and be more strategic about their application. The theory was inspired by two complementary modes of thinking, "fast" (intuition) versus "slow" (deliberation). It defines three key "activity clusters" called LiveCoding Performance, Experimental Interactive Programming, and Exploratory Interactive Programming using the Cognitive Dimensions of Notations as a language to discuss practice-based reflections and user studies of intuitive and deliberative activities in performances, structured experimentation and software development, and open-ended creative exploration. It interrogates the usability trade-offs between the three clusters during the design of LivePrinter, the author's interactive 3D printing (I3DP) and performance system. This positions these activities within the author's practice as an educator, sculptor, performer, and software developer who experiments, composes, and performs, leading to some guidelines for other "creative" IP developers.

INTRODUCTION

Designing a livecoding or interactive programming (IP) system is a "wicked problem" (Rittel and Webber 1973) with many possible solutions. With text-based systems, it is often a process of subtraction where designers choose what parts of the system to hide, abbreviate, or otherwise abstract away in the interests of efficiency and aesthetics. The process is part software engineering, part user experience design, but also part of the artistic practice of the designers – a mix of planning, trial and error, and artistic experiments.

Designers could use some general guidelines that are both simple enough to understand and rich enough to be useful across different modes of development. Ideally, they would be based on empirical studies but attempting them would be "hopeless" (Lewis, Clayton 2017), requiring highly variable and impossibly large data collection activities to capture all the possible uses of a programming system under development, and then categorizing its user types, usage situations, users' previous experiences, and characteristics like age and ability.

Instead, we use a "mechanism schema"-based approach to explain phenomena through descriptions of the *entities, relationships,* and *activities* that narrate how each phenomenon is produced (Machamer, Darden, and Craver 2000). In this type of theory building, based on biological sciences, users and programming systems are *entities* and the Cognitive Dimensions of Notations help establish *relationships* between them. These relationships are described in terms of *activity clusters,* which can be thought of as groupings of similar *activities* that these users (entities) and their systems operate in. Using observations of users and software and expert analysis, we show *why* programmers are using systems and *how* they are useful. This frees us from having to collect enough data to empirically prove that those systems are *inherently* useful for some tasks, or *more* useful than others by comparison (Lewis, Clayton 2017; Russ et al. 2008).

INTERACTIVE PROGRAMMING

Interactive programming can be thought of as a technique for working with software where any part of a program can be compiled and executed at any time: a few characters of text or "nodes" in visual code up to the entirety of the program. The technique is typically used to guide audiences through the audio-visual effects of running code in livecoding performances.

Since the beginning of computer programming in the mid-20th century IP has been used extensively in music and art. In one example, artist Vera Molnar, in her 1975 essay in Leonardo titled "Towards Aesthetic Guidelines for Paintings with the Aid of a Computer", described her "conversational method" as an iterative process of tweaking code and viewing the results on her computer monitor. This form of interactive software development-as-dialogue was essential to her graphical form-finding process (Molnar 1975).

People manipulate code live in other contexts beyond performance, sound production, and audiovisual art. As Tanimoto (2013) observed, they might benefit from "minimizing the latency between a programming action and seeing its effect on program execution". A recent meta-analysis by Rein et al. (2019) identified three main approaches to liveness in software development: "livecoding", "live programming", and "exploratory programming". They described these approaches as incorporating "programming environments and tools that can provide the impression of changing a program while it is running". Here, we refer to all three as IP since they all fundamentally involve interaction and programming, and since there is no other generally accepted term for these related approaches.

These IP approaches have different goals: *creating* new programs (live programming) through immediate feedback on the dynamic behavior of a program as it is being re-programmed or built from scratch; *modifying* existing programs (exploratory programming) to explore the effects of alternate designs when requirements are not fully defined but are yet to be discovered (Sheil 1983; Trenouth 1991) and *performing* code for others and often discarding any artifacts or programs afterward (Rein et al. 2019).

A THEORY OF FAST AND SLOW THINKING IN CODE

This paper proposes a lightweight theory and shared language to help simplify the communication and design of such interactive coding systems aimed at producing "creative" multimedia outcomes, such as generative sculpture, art, music, graphics, and livecoding performances. The theory is based on the three main purposes of IP – program creation, modification, and performance – and focuses on only two main modes of thinking employed during these activities, "fast" (intuitive) and "slow" (deliberate), inspired by studies of human cognition from Kahneman (2011) and Johnson-Laird (1983, 2013). "Speed" of thinking is important because, relative to physical action, performance using programming is a slow, meta-cognitive process of mainly symbolic recognition and composition (e.g. reading and writing text). A designer can use this theory, as they develop and refine an IP system, to reflect on how these "types" of thinking support users (and audiences) to achieve desired outcomes, in different contexts.

The theory defines three "activity clusters" called Experimental Interactive Programming, Exploratory Interactive Programming, and LiveCoding Performance, representing the context-dependent processes of structured experimentation and software development (creating), open-ended, creative exploration (modifying), and performance, respectively. Each activity is supported in a notational system by a unique collection of cognitive design tradeoffs based on the mix of types of thinking involved. These activities range from mostly slow and deliberative (Experimental IP) to quick and intuitive (LiveCoding), with Exploratory IP a mixture of both.

The implementation and implications of this theory are explored in the rest of this paper through the reflections on the initial development and further refinement of the author's LivePrinter system for interactive 3D printing, and their use of the system in their professional practices of education, art, performance, and academic research. The result is some practical guidance for designers of such systems and to inform future studies.

INTERACTIVE 3D PRINTING

3D printing (3DP) is loosely described as a process where people create or otherwise acquire 3D models, load them into special software that renders them into machine instructions for a 3D printer, and then load those instructions into a 3D printer either via the printer itself or special monitoring software, and

finally execute the instructions, extruding melted plastic in precise movements through a hot printer nozzle to render physical objects like boats, cat toys, and door stops. There are other techniques commonly called "3D printing" that use different materials and mechanical systems, but Fused Deposition Modeling (FDM) is the main 3DP technique referred to in this paper.

3DP is a relatively slow process compared to other computer-based activities like working with sound or graphics that are often experienced in or near real-time. Since it builds up objects layer by layer, the speed and size of each layer are important to the overall printing time. Whilst the size of objects is fixed by their design, the height of each layer and thus the number of layers required to build an object is flexible and can be changed in the printing software and physically by using a bigger or smaller printing head on the printer itself. The speed at which the print head moves can also be increased or decreased, leading to longer or shorter print times.

In Interactive 3D Printing (I3DP) systems, such as the author's *LivePrinter,* the emphasis on graphical vs. textual control of the printing process is flipped so that the primary means of 3D printing is through text programming and visualization is secondary and often non-interactive. These systems can be thought of as a new class of IP environments for the real-time control of hybrid digital/physical systems, with a lineage that includes programming for 2D pen plotters and software for choreographing robot movements. Such systems allow the user full control over any aspect of the mechanical system in real time. They prioritize experimentation and exploration with the system over quickly replicating specific objects.

Giving users full control over operations that are interrelated in complex, non-linear ways has usability drawbacks. Describing even simple objects textually in the GCode language, a popular and relatively simple syntax for encoding machine operations, can take hundreds or even thousands of lines of code. IP in GCode is possible, but painfully slow. This is one reason why the *LivePrinter* system for programming 3D printers was created – to create a library of functions in a procedural language (JavaScript) so that 3D objects can be described procedurally and even algorithmically in parts and still render to universal GCode, including complex operations like "priming the material" that include multiple lines of GCode. Since the start of this project, other systems have appeared, like the FullControlXYZ project[1].

FAST AND SLOW CODING WITH MACHINES AND HOT PLASTIC

Speed is important when working interactively with 3D printers because the hot material is always dripping out of the print head. When movements are too slow, too much drips out and further printing operations will be stringy, leaving gaps in the printed model, or the material will ball up and start to burn. Speed is also important when performing code in front of an audience, as with livecoding, to keep the audience engaged as the performer controls the tempo and duration of the performance. Since *textual* livecoding is arguably a mostly cognitive activity that relies more on mental operations than physical dexterity (Sayer 2015), the ability of a performer to think in code is the main limiting factor in their performance — the speed limit governing how fast they read code, change code, and write new code.

A long-standing tradition from at least Spinoza through William James to the present consists of splitting cognition into two processes in a so-called "Dual Process theory of the mind." This was popularized by Kahneman (2011) and has similarities to the "Mental models" theory from Johnson-Laird (1983, 2013). Both identify two distinct cognitive processes of intuition (sometimes called "instinct") and deliberation (or "logical thought"). Whether these theories are scientifically valid because of intrinsic statistical errors has been the subject of debate, including from Kahneman, but the concept of "fast and slow thinking" is still potentially useful to reflect on when designing IP systems, especially those for livecoding performances.

"Intuition" relies on the quick retrieval of knowledge stored in people's long-term memory. It likely happens at a sub-conscious level and is a faster process than "deliberation," or conscious and explicit thought. Deliberation relies on working memory, which is capable of problem-solving but limited in resources such as memory and processing bandwidth. Intuition is faster, but its speed comes at a cost: it

[1] https://github.com/FullControlXYZ/fullcontrol

is based on previously learned heuristics, is slow to update, and has little to no access to conscious processes of thought and working memory. As such, intuitive thought processes are incapable of doing even simple arithmetic, such as counting (DeStefano and LeFevre 2004; Sweller 2003; Johnson-Laird 2013).

Any attempt at switching models and looking at a situation "from different points of view" implies a more deliberate way of thinking. Deliberation is a much more involved and slower cognitive process capable of recursion, searching for and applying alternative models, and more complex arithmetic including calculating probabilities. As Johnson-Laird (2013) puts it, "the distinction between intuition and deliberation is in computational power: intuitions are not recursive, but deliberations can be."

THE COGNITIVE DIMENSIONS OF NOTATIONS

The Cognitive Dimensions of Notations Framework (CDNs) is a list of "cognitive artifacts" that has often been used as an evaluation and discussion tool for interactive systems and their notation (Green and Petre 1996; A. Blackwell 2005; A. Blackwell et al. 2001). The CDNs are not meant to be an exhaustive checklist of potential usability issues but a vocabulary to help designers who are creating activity-specific interactive systems to understand some basic dimensions of usability, and to balance these dimensional effects against one another whilst supporting those activities.

There are 14 current CDNs that have been revised and added to since their introduction by Green and Petre in 1996. They are designed so that dimensions can be paired up and varied in their relation to one another (inverse, direct, none) whilst looking at their effects on others. For example, "Hidden dependencies" occur when a system's notation refers to a relationship that isn't visible to the user of, or participant in, that system. Hidden dependencies could be made visible by adding them to the screen, however, this could increase the cognitive load of a person who must read and make sense of more text or graphics. So, the CDN "hidden dependencies" may have an inverse relationship to the CDN "visibility".

Importantly, the CDNs are meant to exist in the context of specific activities that users will be tasked with when working in a particular environment. The main activities that have evolved out of the CDNs over the years are "incrementation" (adding new notation), "transcription" (copying or translating from one notation format to another), "modification" (changing notation), "searching" (looking for notation in an interface), plus "exploratory design" (open-ended making activities akin to sketching) and "exploratory understanding" (understanding the structure of concept or the reasoning behind its construction). Changing activities will likely change the balance of the desired traits, such as designing for highly structured user activities like spreadsheet editing which rely on data entry, data modification, and copying versus programming environments that support more free-form exploration that requires less structural constraints.

On top of these 14 or more dimensions and six user activities, other researchers have developed "patterns of user experience" that attempt to group user activities into contextually specific "patterns of use" such as with "performance programming" like livecoding and its various uses in "creating a narrative" (Alan Blackwell 2015; Alan Blackwell and Fincher 2010). Taken together, all this high-level terminology is challenging for a designer to come to grips with when designing a new IP system. The novel "fast/slow" theory suggested in this paper consolidates these theories and practices using descriptions of how certain subsets of CDNs relate to each other inside the ExperIP, ExploIP, Livecoding clusters, along with expected user activities.

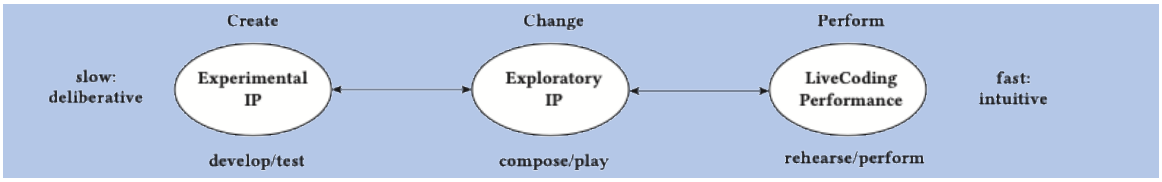

Figure 1: Activity clusters ranging from deliberative to intuitive, with examples of context at the bottom of each.

METHODOLOGY

The following section follows a mechanistic approach to theory-building (Machamer, Darden, and Craver 2000; Russ et al. 2008) based initially on first-person reflections by the author interrogating the experience of the activity clusters of Livecoding Performance, ExploIP, and ExperIP in their artistic practice and in preparation for a series of live performances, culminating in the performances themselves. The CDNs are used as a shared language to help pick apart complex relationships between user experiences and programming system features in each cluster. The author's development work, experiments, and performances are part of their broad practice as a computational artist creating video and sculptural objects based on musical concepts and programming, organizer of live events, an educator running a Computer Science course at a university, and teaching workshops around the world, and as an occasional consultant developing software for private clients.

The experiments with interactive 3D printing include the development process for LivePrinter from 2018 until 2022, and its refinement of a performance system from 2022 until September 2024 (at the time of this writing). This includes a series of 6 workshops with a broad range of participants from different backgrounds, some with no experience with coding, some with little to none with 3D printing, some with extensive 3D printing experience, a few self-described veteran livecoders and undergraduate and graduate students working in computer science and computational art. Data was collected from them in the form of informal interviews during and after the workshops, questionnaires with space for free response, and a reflective diary from the author acting as the main facilitator. These were carried out during the early development stage of LivePrinter from 2018-2019 in London, New York, and at ICLC 2018.

During later development from 2023-2024, LivePrinter evolved to use musical notation to describe sculptural forms that the author was interested in. This period was documented through the production of a series of videos[2], physical sculptures, code snapshots from LivePrinter, Git repository history from specific sessions and an ongoing reflective diary kept online[3].

The performances in 2024 were associated with the LiveCode London collective of livecoders and A/V artists and the Creative Computing Institute (CCI) at the University of the Arts, London. They were co-organized by the author as well, and took place in various venues: an Algorave at an intimate, underground music venue and bar called The Crypt known for live jazz nights, in Camberwell, London, in front of about 130 people in early June 2024; an AlgoEvent (Algorave + workshops) at a relatively large space in the old Limehouse Town Hall, London, in front of about 80 people in late August 2024; at the CCI on two dates for returning students and staff in late September 2024. They were documented in photographs and video from the events, reflections on the personal experience of the author, and their conversations with other performers and participants recorded in the reflective diary.

CDN	**Example(s)**	**In Intuitive**	**In Deliberative**
Abstractions	High-level functions; external code libraries	High	Low
Consistency	Following naming conventions for different printer movements (travel, printing) for faster guessing	High	Low
Diffuseness	Long comments in code; multi-line code instead of shortcuts; long variable names	Low	High
Error-proneness	Referencing functions or variables that don't exist; accidentally moving the print head so it crushes an object	Low	High
Hard mental operations	Calculating where to move the print head to print a side of a polygon	Low	High

[2] publicly available on the author's YouTube channel at https://is.gd/kcxiO4

[3] These are available by contacting the author.

Hidden dependencies	Trying to remember if the material has been retracted or not; changing a parameter like layer height that affects how a shape is being drawn in unexpected ways	Low	Low
Premature commitment	doing things in order, like setting up a sequence of notes in a melody and starting them playing	High	Low
Progressive evaluation	Current progress and state can be checked: UI elements including automatic numerical feedback and a visual log of events, accessible programmatically	Low	High
Provisionality	Committing to a specific narrative series of events in a performance and following them through	High	Low
Role expressiveness	Function and variable names that aren't clear and can't be guessed can break the flow in performances; roles in experimentation are often in flux, however	High	Low
Secondary notation	Code comments and sheets of paper to draw set lists, diagrams, and take notes	Same	Same
Viscosity	How hard it is for a programmer to change parts of the system and get a result, either through executing code or clicking through the interface: high viscosity means key functions and variables, like 3D path generators, cannot be changed easily or at all	High	Low
Visibility	The more information and notation visible on screen, the easier it is to work on complex coding tasks like function library development	Low	High

Table 1: The 13 CDNs used in this study, along with examples of their conception concerning Interactive 3D Printing and their optimal level in deliberative vs. intuitive activities.

DESCRIBING EXPERIMENTAL IP

In an ExperIP activity, programmers are often working individually on self-guided tasks that are purpose-driven and have some notion of success or failure. These can be likened to experiments, even if they aren't as rigorously constructed as scientific or engineering experiments. Programmers may write software tests to see if features are working as expected; they may experiment with potential solutions to problems of form, shape, and musicality that involve writing complex code that will never be directly visible during performances such as libraries of geometric functions; or try alternative solutions to their interface design issues, even extending and re-writing the IP system itself.

Not all this work will be aimed at producing novelty – they may also spend time transcribing, translating, and modifying algorithms or mathematical expressions from published research or other libraries and languages into usable code for their system. By their nature, these are slow, deliberative tasks that require thinking and reflection. Programmers may aspire to complete these tasks as quickly as possible, but their experiments may still take hours, days, or even years to achieve results that they can use in other activities, like performances, or for creating artifacts such as videos, audio recordings, and physical objects.

"Making useful errors" in a test-driven development process is a way to colloquially describe the main experience of ExperIP since so much of the work is in building new functionality, finding the parts of the system that don't work, and trying out new techniques in forms and musical ideas that often result in terrible aesthetics. Key to this process is keeping what happens in the system visible to the programmer so that they can identify the mechanisms of helpful errors and incorporate them into their later work.

Assuming that it is mainly a deliberative way of working, ExperIP can be described using a combination of the user activities "exploratory design" and "exploratory understanding", since programmers are expected to be both creating new things and then interrogating them iteratively. In exploratory design, the literature has established that designers will make a range of minor modifications, refactorings, and even fundamental changes to software architecture. As Blackwell (2005) wrote, "Viscosity has to be as low as possible, premature commitment needs to be reduced, visibility must be high, and role-

expressiveness – understanding what the entities do – must be high." This is also expected of the key CDNs that make up ExperIP:

1. Low "consistency": often not very because of the intermediate state of development: since things change, the role of notation and its purpose may change and break things
2. Many "hard mental operations": often many, because clarity usually comes with work and experience
3. High "diffuseness": as in quite diffuse since code is a form of "talking things through" and self-conversation
4. High "role expressiveness": understanding what does what and how is important
5. High "premature commitment": experimenting with an optimal or preferred order for writing and executing code
6. High "provisionality": commitment to a course of action — trying different possibilities at a lower cost and then later deciding which to commit to
7. High "viscosity": more information (diffuseness), more ways of doing things, and higher visibility means less viscosity since programmers can change almost anything
8. High "visibility" and "progressive evaluation": it is essential to be able to view the current state of the system and its query its component parts during operations

Case study of Experimental IP with LivePrinter

ExperIP describes the core development of *LivePrinter*: a slow, iterative, and error-filled process of multiple successes and failures, using development environments with different strengths and weaknesses. The system under development included a backend server that communicated with the printer and a front-end interactive code editor for users. Development on the IP system had to be done in another editor since LivePrinter could not modify itself in any permanent fashion on the hard drive, only temporarily in the memory of the web browser. This process was a blend of IP and whole program compilation since the system had to be compiled before the interactive editor could be loaded and used. The interactive system was then used for sketching out functions and variables that often found their way into the backend code.

Without a clear roadmap for developing an experimental 3DP system, high "provisionality" was important so code experiments in the back and front end, such as new movement functions and exposed printer variables, could be attempted quickly and then iterated. High "visibility" was also key because of the need to see and understand the 3D printing system and Marlin firmware that powered it during experiments, and because the rapid pace of development meant that the system was often not very "consistent" when trialing different naming conventions for functions and variables to see which worked best in practice. This meant a very high "viscosity", especially since multiple code editors with multiple files were often open, as well as in-code documentation and external documentation on websites ("secondary notation").

There were many "hard mental operations" for the developer since it was difficult to design LivePrinter functions to support users working in less "diffuse" ways in a more Exploratory or Livecoding activity, especially handling material flow and the geometry of printer movements. For example, the *retraction* process for materials handling was, and still is, a difficult one because it is a technically named, non-intuitive concept with low "role-expressiveness", low visibility, and high commitment that is prone to errors. The molten material is often "retracted" or pulled backward up into the print head a certain amount (varying with printer models and materials) after a printing operation to prevent it from dripping and losing the material needed for the next print operation. This action is hidden and often unpredictable in practice.

Another example shows how ExperIP is used for both creating and modifying software, especially code refactoring and modularization where parts of a software program can be split out based on their functionality. This was seen in 2023-2024 when development focused on making livecoding less of a cognitively intensive activity by providing 2D and 3D repeating paths that could be more easily

manipulated in live performances, with less diffuse code and less hard mental operations for the performer. These paths would be mathematically and procedurally generated, providing functions and parameters for livecoding to get slightly different results in each performance.

The activity began with hand-drawn sketches ("secondary notation") for repeating, zig-zag paths which were partially implemented in LivePrinter's interactive editor in a back-and-forth process from sketching-to-interactive-coding-to-sketching for over a year before they became too complicated and time-consuming to run in the line-by-line interactive editor. During that time, some of the best shapes were made because of code "errors" made in the moment of performance. These shapes were first envisaged as zig-zag paths in 2D space, stacked into 3D layers, with each point in the paths modified by a combination of sinusoidal functions that could vary over time.

What made them interesting was a miscalculation in the period of the oscillations of the sinusoidal shapes that didn't correspond to the number of points in the paths as intended but produced multiple oscillations per layer. These new families of shapes and performance concepts proved to be quite useful for making complex shapes with hollow cavities without using more complex logic like the "Indeterminate M" and "W" series documented in the paper "Space as Time" (Raskob 2023). The original "mistakes" were time-consuming to analyze and reconstruct, but eventually, the path functions were consolidated into an external library (GridLib) that could be loaded into LivePrinter via editor commands during performances.

Testing and deploying this new library required changes to the entire LivePrinter codebase to make it more modular. Then, the difficulty of visualizing and understanding the paths spawned another library (VisLib) aimed at rendering LivePrinter operations and outcomes in a 3D preview, on a step-by-step basis, so programmers could visualize the system without committing to a time-consuming 3D printing operation.

This demonstrates different development activities reinforce one another, with ExperIP spawning ExploIP and LiveCoding tools providing high "visibility" for paths that users can take to decrease their "premature commitment" when choosing one, giving them the flexibility to "provisionally" try others.

LIVECODING

Livecoding should be the quickest and most intuitive of the three clusters since it is time-sensitive and intuition is the quickest mode of thinking, according to cognitive theories. An assumption inherent in this cluster is that the performer is an expert, or near-enough one. That implies familiarity with the livecoding system and a history of practice and rehearsal. Performances may be improvised, but the performer usually has some understanding of the effects of their code before it executes. If they were novices or in the process of learning, their activities would fall into one of the other more deliberate clusters, depending on their level of familiarity and programming skills. Additionally, in a livecoding performance mode where the interface is projected for an audience, the demands of the audience for maximum visibility (and legibility) are at odds with the performer who must concentrate on quickly operating the system (Bruun 2013; Wakefield et al. 2014).

The risk is not of writing software that will fail tests, or failing experiments, but "… the risk of lacking aesthetic merit: for instance, the risk to be boring or incomprehensible and meaningless. It is the same risk all artists face in their respective fields." (Bertinetto 2013, p.9). Thus, failure in livecoding is ultimately aesthetic, rather than strictly technical, since a series of technical failures can still be of aesthetic interest to an audience, allowing them to "revel in the glitch or crash, as a demonstration of fallibility or fragility of the machine" (Blackwell 2015). Users are less concerned with writing software artifacts than performing programming in front of audiences, mainly combining the CDN activities of *incrementation*, *modification*, and *searching* in the interest of speed and a pre-planned performative flow because "the range of possibilities of a right performance is quite narrow" (Bertinetto 2013, p.9). Advanced

practitioners might take time to write new functions leading toward new improvisations, but it is rare for them to write complex data structures or algorithms live (Blackwell 2015).

In the interest of speed, the system should offer the user "terse" (as opposed to "diffuse") syntax and UI controls for executing complex processes with minimal cognitive effort in the moment of performance. That still means reading, understanding, and remembering them during a performance, but with less emphasis on remembering since working memory is limited and it is often easier to find similar code and modify it than to start from a blank screen. However, making "visible" that syntax comes at the expense of a considerable increase in the overall "diffuseness" of text and graphical notational elements, which is an issue for an already information-dense, verbose text editor. Adding elements takes up precious screen space and increases the cognitive bandwidth for users to find and focus on them, or to be able to recognize and skip over them and focus on other elements.

Thus, the expected relationship of the CDNs that make up the intuitive LiveCoding Performance cluster should be somewhat opposite to the more deliberative ExperIP cluster:

1. High "consistency": syntax should be consistent and intuitively guessable
2. Low "hard mental operations": more quick actions with less thought so as not to cognitively overload the performer
3. Low "diffuseness": less diffuseness because of the time it takes to read and type
4. High "role expressiveness": again, understanding what does what and how is important but this is often overridden by the need for brevity (speed) and dramatic purpose
5. High "premature commitment": decisions need to be made in the moment, even if they aren't fully understood
6. Low "provisionality": the performer should commit to a course of action and stick with it, with little time to deviate and experiment without too much risk of failure (meaning boredom)
7. Low "viscosity": less information (terseness), fewer possible ways of doing things, and lower visibility means more viscosity since programmers can't easily access parts of the system
8. Low "visibility" and "progressive evaluation": less, since execution and follow-through are more important than viewing parts of the system and its current state

Case study of Livecoding with LivePrinter

Brevity was a goal of LivePrinter's editor, which developed from the author's experience of livecoding performances in front of audiences and struggling to type and then match up complicated syntactical elements like pairs of curly braces and square brackets. To solve this, the LivePrinter "minigrammer", inspired by the TidalCycles's use of "chains" of functions and lists of patterns, shortened JavaScript by stripping out punctuation, brackets, and asynchronous function calls.

For example, the JavaScript to retract the print material, raise the print head quickly and then move the print head:

```
await lp.retract();
lp.speed(50);
await lp.up(40);
await lp.moveto({x:50, y:50});
```

The same, using the minigrammar:

```
# retract | speed 50 | up 50 | mov2 x:50 y:50
```

More of these high-level "abstractions" such as variables and functions help to decrease overall "visibility" by consolidating concepts and functionality into fewer choices of syntax which are harder to modify (increased "viscosity"). This lowers "provisionality" because programmers have less control over how they can deploy these abstractions. They also may *decrease* other types of "viscosity" such as "repetition viscosity" because programmers can get more results with fewer steps and less typing.

These powerful abstractions come at a cognitive cost for remembering what these abstractions mean and do. As Kutar and Basden (2010) observed, the CDN "abstraction" can be redefined in cognitive terms as "an effective substitution between reality and the human mind", which makes an effective level of abstraction relative to the experience of a particular user. An expert will have a much higher threshold of

abstraction concepts than a beginner – it is well established in computer science literature that people learning to program must overcome certain cognitively difficult "threshold concepts" like loops and recursion before they can effectively use a programming language.

Terms and syntax for the I3DP-specific notation often required deeper knowledge about how the functions were internally created and used. The programmer was forced into a deliberative mode of investigation into those "hidden dependencies" that might involve looking through documentation or even source code. For example, during practice sessions the author often reverted to a more ExploIP mode when a creative idea forced them to check the source code files or extend the system's functionality, demonstrating that the "viscosity" of the system is strongly related to the "abstraction-level" of its syntax.

Part of the author's practice with GridLib and LivePrinter consisted of working in Experimental and ExploIP activities to create livecoding-specific presets with parameters to tweak in performances. The black-box natures of these presets reduced the "visibility" of what went into those processes, meaning that "hidden dependencies" were created and the performer was not able to effectively "progressively evaluate" or keep track of what was happening, embracing the drama of "reveling in the glitch or crash".

This happened in the fourth performance, where the print head became misaligned when it hit an object and on the bed. The performer managed to successfully recalibrate the bed so quickly that the audience was mostly unaware, (from later conversations with audience members). In the second performance, the performer failed to recalibrate the bed, forcing them to run through the preset sequences even though the shapes would be printed too high above the print surface, making them messy and unpredictable. They tried changing simple parameters in code such as lowering the print speed and raising the temperature, since that often produces more precise shapes and hotter material bonds netter to gaps in surfaces. The first shape recovered its intended form over time, but the second one ended as an untidy ball. The audience and performers stared at it in silence as it dangled and when it fell and they broke into applause, celebrating a glitch that literally produced a crash.

This experience encapsulates the experience of 3DP quite well – the low "visibility" of the print bed state and printing operations in general, the very high "viscosity" of the bed-leveling functionality during performances, plus the "error-proneness" of the precision mechanisms getting out of alignment.

EXPLORATORY IP

ExploIP is is and deliberative Experimental ways of working where the goal is to modify and extend a system, as in composing a new performance or working with new functionality that might find its way back into the core system in a future ExperIP activity. In this cluster, we expect programmers to swap between modes of improvisatory understanding and quick, sketch-like experimentation. It is a more forensic, reflective mode that prioritizes the "provisionality" of trying out different options, balanced with "in-the-moment" reflections about which option is aesthetically successful:Users quickly and playfully sketching ideas, working in a "thinking-through-making" mode (Schön 1991) where they can use existing code functionality to get results without too many "hard mental operations" or too many "hidden dependencies". ExploIP leverages the activities of "incrementation", "modification", and "searching" for getting relatively quick results without writing much code or needing to look through too much documentation, with some "transcription" activities of copying, modifying or translating an algorithm from a similar piece of code. Speediness is still a design goal to keep sessions short and playful enough to be achievable, but less so than with performance. the activities of "incrementation", "modification", and "searching" for

The relevant CDN relationships for the "in-between" mode of ExploIP would be expected to be:

1. Balanced/low "hard mental operations": a balance of quick actions so as not to cognitively overload the programmer with access to other modes of "secondary notion" like sketches and diagrams to allow for more innovative explorations
2. High "role expressiveness": high, because understanding what does what and how is important
3. Low "premature commitment": as demonstrated in user studies and in practice, people can learn to deal with fairly complex orders of operations if they are repetitive and expected

4. High “provisionality”: there should be space and time to deviate and experiment without too much risk of failure
5. Balanced “viscosity” and “visibility”: there should be a diverse range of ways of doing things, with enough visibility to allow programmers to access most parts of the system, even if they need to use other applications to do it
6. Balanced “visibility” and “progressive evaluation”: a balance, since programmers need to understand the results of their actions, although not to the same depth as with tests and experiments

Case study with LivePrinter

An important strategy was to focus on ExploIP over ExperIP working, ignoring errors and buggy code in the service of sketching new ideas. This was helpful in the user studies where retraction functions were error-prone and participants got confused about why no material was coming out of the print head. When told about the bug, they were able to ignore it and focus more on being “fun” or “playful”, creating “blobs” and “nests” instead of precision shapes.

The author used ExploIP when composing new functions, trying out presets, and reflecting on variations of previous work, often recording videos or screenshots to review at the start of sessions. Sometimes they referred to a sketchbook to make drawings or used another piece of software to create 3D to test out a working with the I3DP system. This constant context-shifting from notation systems was helpful for working through “hard mental operations” outside of the system’s limited syntax and “visibility”, but it also distracted from the main task at hand if relied upon too often.

Sometimes the system was too buggy or lacked key functions needed to realise the author’s ideas – not unlike Sam Aaron’s experience as documented in Blackwell and Aaron’s ‘Craft Practices of Live Coding Language Design’ (2015). This meant swapping to an ExperIP mode to work with Visual Studio Code, working on the GridLib library or core API for LivePrinter.

CONCLUSIONS: DESIGN RECOMMENDATIONS

These explanations of the mechanisms of each activity cluster lead us to some recommendations for designers of such systems that can be explored in future studies.For ExperIP, where the goal is to create usable software with new and unpredictable uses, we should expect to make mistakes and encounter problems because trial and error and “happy accidents” are part of the process. Perhaps we can embrace more of a sense of playfulness and the fun inherent in this method of working instead of writing tests and trying to avoid errors. Experiments probably cross over between multiple tools and environments and will likely involve re-writing or extending the interactive system, then working with the interactive system to test the results and create sketches for future development.

Designers of ExperIP systems should consider the following advice:

1. Don’t be too strict enforcing consistent naming and coding conventions, allow users to write loosely and diffusely. Trust that code will eventually settle into understandable patterns, if it produces worthwhile results.
2. Allow for multiple ways of executing code and seeing the results. In a deliberate (exploratory or experimental) activity, the programmer might want to play around with different sequences for doing things without committing to any course of action, and then later reflect on what the optimal or preferred order was.
3. Allow programmers to work with complex interfaces that have multiple elements, even if they are sometimes hard to understand. Let them work with multiple views, like 3D previews. Don’t hide things, even if that means more work on their part thinking through their intentions. This may mean splitting a project into multiple interfaces, each supporting one or more of the three activities. Trust that simplicity will come when they switch to more intuitive, performative modes of working.
4. Try to understand sources of error and even how to replicate them by making as much of the system visible as possible, either passively through a static interface or actively through logging and console functions.

A LiveCoding IP activity is more about performing in front of others: quickly writing code for people to focus on, that fits a particular performance narrative. "Interestingness" is the most important criteria, which implies the performer should be always visibly doing something and not inwardly deliberating over their next action or pausing to debug code (unless that is part of the performance). We expect programmers to work fully inside the IP system, without switching between other environments or tools. As such, recommendations are:

1. Give the programmer mechanisms for writing terse code so they can type quickly but try not to enforce naming conventions outside of a basic consistency of guessable and easily rememberable notation so that they are free to express themselves as needed.
2. Keep options limited: only provide high-level abstractions with clear use cases.
3. Focus on what can quickly be done, not what can be extended: don't worry about providing different mechanisms for getting different results, give programmers clear paths to commit to and build upon.
4. Prioritise simplicity by showing less information beyond code and provide multi-purpose feedback areas (like a console or logging section) instead of more detailed debugging tools to avoid distraction
5. Allow for secondary notation in the form of code comments, notes, or decoration so that programmers and the audience can have cognitive bookmarks to help them quickly navigate through code blocks.

An ExploIP mode of working is akin to a performer practicing fundamentals or for their next engagement, or a designer sketching out new possibilities. Like ExperIP, ExploIP sessions will likely cross over multiple tools but with a focus on understanding and reference to the underlying system, rather than extending and rewriting features of it. Towards these ends, we expect playful experimentation without reaching the depths of ExperIP, more in trying out a feature than rigorously testing the implementation of that feature. The design focus is on providing a balance between control and visibility, and between commitment and choice of action. In the author's studies, the programmer was always using a combination of LiveCoding and Experimental systems to achieve Exploratory activities since no one system supports this activity. Whether it a separate "view" or editor mode is useful for this activity alone needs to be studied, but we can provide some guidelines for working in this mode regardless:

1. Prioritise playfulness and fun over simplicity and depth
2. Allow for different views of the underlying system and the IP interface(s): balance the visibility of relevant documentation and source code with a focus on a smaller range of creative possibilities to explore to keep from getting overwhelmed
3. Support minor edits and clarifications to the core IP system, like get/set functions and fixes to names and variables for consistency and increased role expressiveness
4. Develop new abstractions as needed but also be ready to cull unnecessary ones
5. Stay relatively organized and consistent with syntactical conventions so actions can be traced back later and replicated or reflected on
6. Keep secondary documentation handy – comments, drawings, diagrams – and keep adding/subtracting/clarifying them in the process.

REFERENCES

Bertinetto, Alessandro. 2013. 'What Do We Know through Improvisation?' *Disturbis*, no. 14, 1–22.

Blackwell, A. 2005. 'Cognitive Dimensions of Notations'. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 3–3. IEEE.

Blackwell, A., C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, et al. 2001. 'Cognitive Dimensions of Notations: Design Tools for Cognitive Technology'. In *Cognitive Technology: Instruments of Mind*, edited by Meurig Beynon, Chrystopher L. Nehaniv, and Kerstin Dautenhahn, 325–41. Berlin, Heidelberg: Springer Berlin Heidelberg.

Blackwell, Alan. 2015. 'Patterns of User Experience in Performance Programming'. In *Proceedings of the First International Conference on Live Coding*, 12–22. Leeds, United Kingdom: ICSRiM, University of Leeds. https://doi.org/10.5281/zenodo.19315.

Blackwell, Alan, and Sally Fincher. 2010. 'PUX: Patterns of User Experience'. *Interactions* 17 (2): 27–31.

Bruun, K. 2013. ‘Skal vi Danse Til Koden? Kunsten.Nu’. 18 November 2013. http://www.kunsten.nu/artikler/artikel.php?slub+livekodning+performance+kunsthal+aarhus+dave+griffiths+alex+mclean+algorave.

DeStefano, Diana, and Jo-Anne LeFevre. 2004. ‘The Role of Working Memory in Mental Arithmetic’. *European Journal of Cognitive Psychology* 16 (3): 353–86.

Green, T. R. G., and M. Petre. 1996. ‘Usability Analysis of Visual Programming Environments: A “Cognitive Dimensions” Framework’. *Journal of Visual Languages and Computing* 7 (2): 131–74.

Johnson-Laird, Philip N. 1983. *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness*. 6. Harvard University Press.

———. 2013. ‘Mental Models and Cognitive Change’. *Journal of Cognitive Psychology* 25 (2): 131–38. https://doi.org/10.1080/20445911.2012.759935.

Kahneman, Daniel. 2011. *Thinking, Fast and Slow*. Macmillan.

Kutar, Maria, and Andrew Basden. 2010. ‘Towards a Cognitive Model of Interaction with Notations’. In *Advances in Cognitive Systems*, 415–34. IET Digital Library. https://doi.org/10.1049/PBCE071E_ch15.

Lewis, Clayton. 2017. ‘Methods in User Oriented Design of Programming Languages’. In . 28.

Machamer, Peter, Lindley Darden, and Carl F. Craver. 2000. ‘Thinking about Mechanisms’. *Philosophy of Science* 67 (1): 1–25. https://doi.org/10.1086/392759.

Molnar, Vera. 1975. ‘Toward Aesthetic Guidelines for Paintings with the Aid of a Computer’. *Leonardo* 8 (3): 185–89. https://doi.org/10.2307/1573236.

Raskob, Evan S. 2023. ‘Space as Time.’ *Trópos* Vol. 15 No. 1 (November):177-198 Pages. https://doi.org/10.13135/2036-542X/9040.

Rein, Patrick, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. ‘Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness’. *The Art, Science, and Engineering of Programming* 3 (1). https://doi.g/10.22152/programming-journal.org/2019/3/1.

Rittel, Horst W. J., and Melvin M. Webber. 1973. ‘Dilemmas in a General Theory of Planning’. *Policy Sciences* 4 (2): 155–69. https://doi.org/doi:10.1007/BF01405730.

Russ, Rosemary S., Rachel E. Scherr, David Hammer, and Jamie Mikeska. 2008. ‘Recognizing Mechanistic Reasoning in Student Scientific Inquiry: A Framework for Discourse Analysis Developed from Philosophy of Science’. *Science Education* 92 (3): 499–525. https://doi.org/10.1002/sce.20264.

Sayer, Tim. 2015. ‘Cognition and Improvisation: Some Implications for Live Coding’. In *Proceedings of the First International Conference on Live Coding (ICLC) 2015*, 87–92. Leeds, UK: ICSRiM, University of Leeds. https://doi.org/10.5281/zenodo.19328.

Schön, Donald A. 1991. *The Reflective Practitioner : How Professionals Think in Action*. Farnham: Ashgate.

Sheil, Beau. 1983. ‘Power Tools for Programmers’. Datamation Magazine. 1983.

Sweller, J. 2003. ‘Evolution of Human Cognitive Architecture’. Edited by B. Ross. *The Psychology of Learning and Motivation* 43:215–66.

Tanimoto, Steven L. 2013. ‘A Perspective on the Evolution of Live Programming’. In *Proceedings of the 1st International Workshop on Live Programming*, 31–34. LIVE ’13. Piscataway, NJ, USA: IEEE Press. https://doi.org/10.1109/LIVE.2013.6617346.

Trenouth, J. 1991. ‘A Survey of Exploratory Software Development’. *The Computer Journal* 34 (2): 153–63. https://doi.org/10.1093/comjnl/34.2.153.

Wakefield, Graham, Charles Roberts, Matthew Wright, Timothy Wood, and Karls Yerkes. 2014. ‘Collaborative Live-Coding with an Immersive Instrument’. In *Proceedings of the Conference on New Interfaces for Musical Expression (NIME) 2014*, 505–8. nime.org. http://www.nime.org/proceedings/2014/nime2014_328.pdf.